03 February 2025 19:19

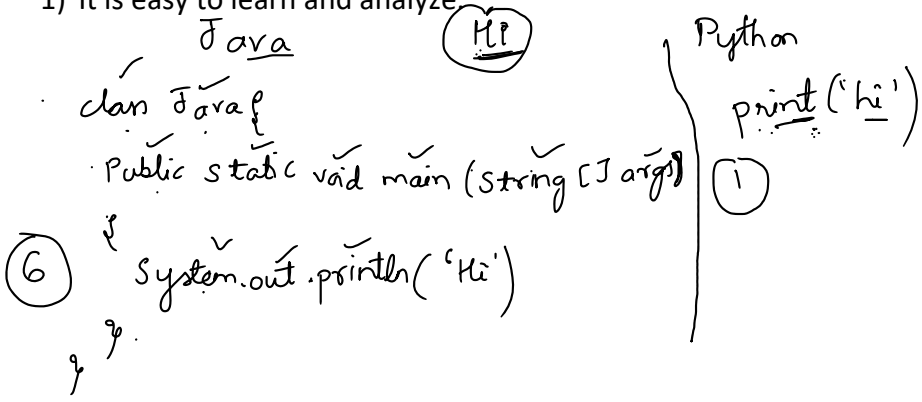
Day-1

Python: It is a programming language .

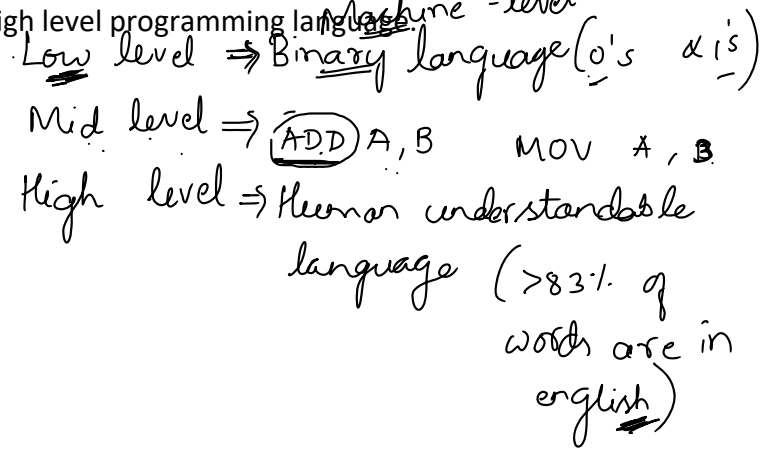
Guido Van Rossum is the one who created Python in the year 1991.

Features of Python:

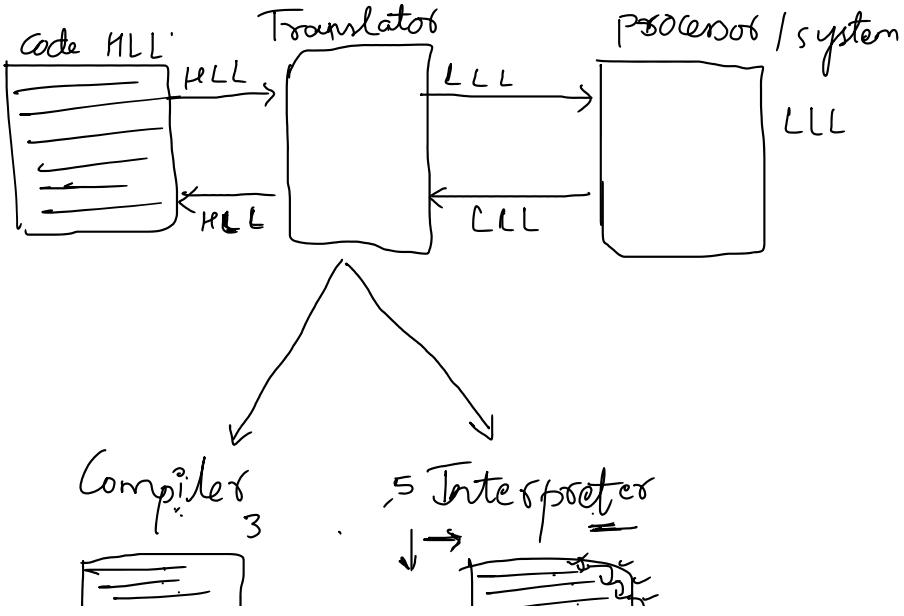
1) It is easy to learn and analyze.

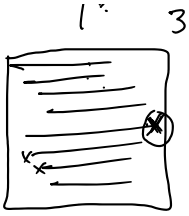


2) It is high level programming language.



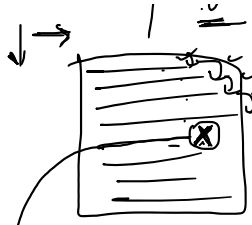
1) It is interpreted language.





output at a time

number of error present



stop the execution

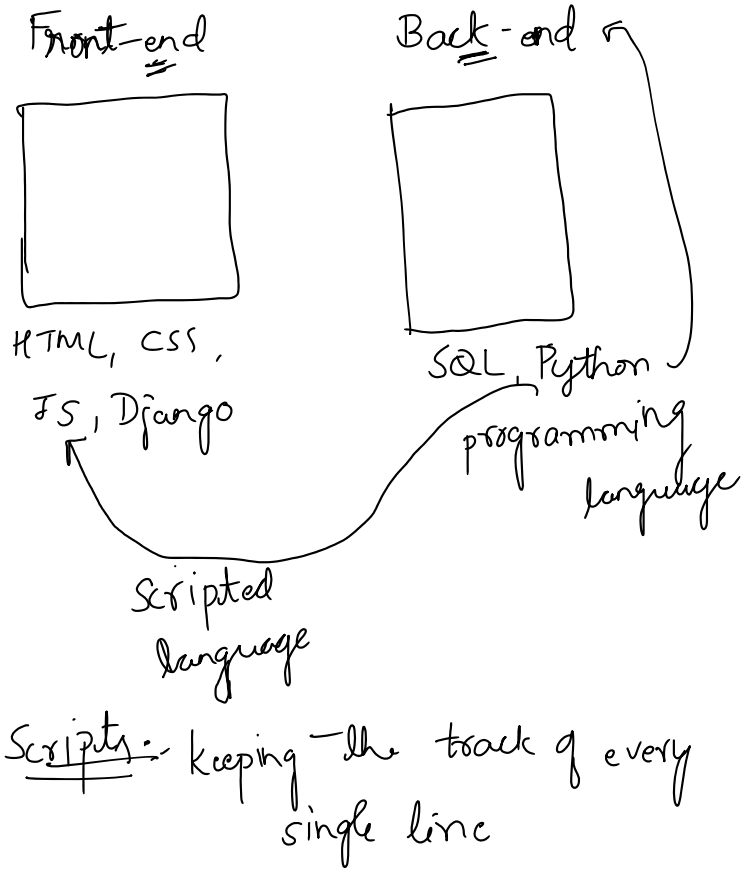
line by line execution

exact line where error occured

Which is more efficient?

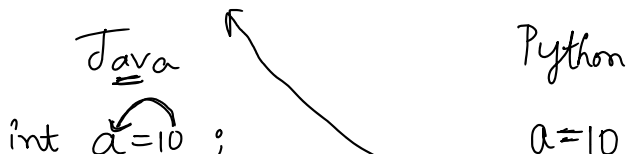
⇒ Compiler

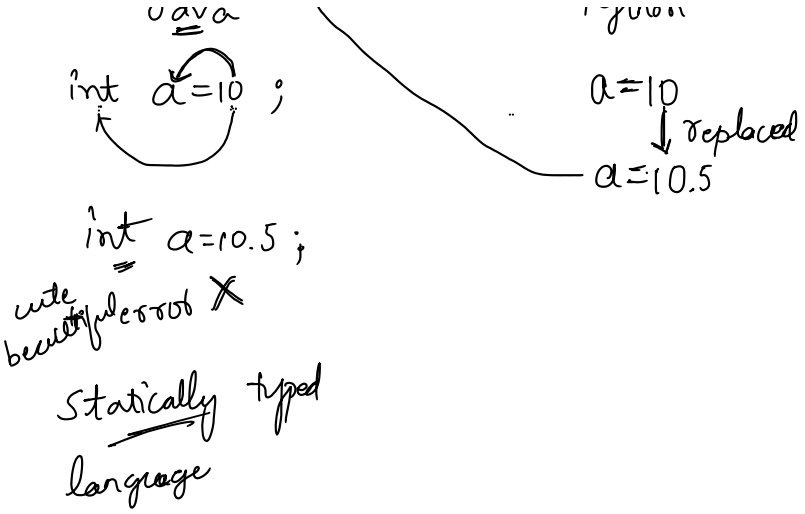
4) It is scripted language.



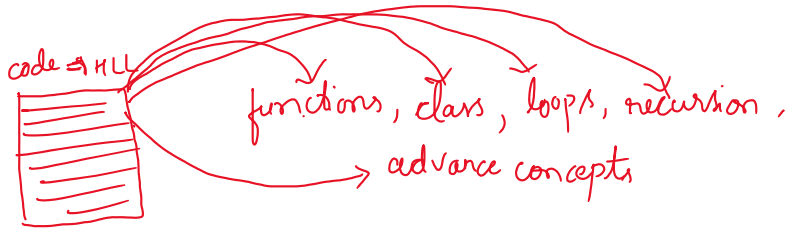
Scripts:- keeping the track of every single line

5) It is dynamically typed language.





6) It is multi paradigm.
---> A single program can be written in multiple formats.

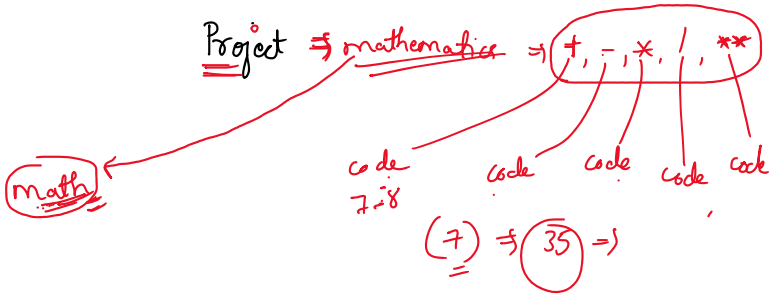


7) It supports OOPS concept.
---> OOPS - Object Oriented Programming system.

8) It is open source.
---> We can easily download python software without paying single rupee.
We can also contribute code for python.

Numpy, Pandas are contributed by developers.

9) It supports huge libraries.
--->



Python has 7+ Crore Libraries.

10) It is platform Independent.
---> platform - Operating System.

What is python?
---> It is a general purpose, high level programming language.

Day-3

Library Function:
---> It is a function which is predefined by the developer to perform some particular task.

Note:
-> We can access the features of Library Function but we cannot modify it.

Types in Library Function:
- Keywords
- Special symbols/Operators
- Inbuilt function.

1) Keyword:
---> They are universally standard words, which is predefined by the developer to perform some particular task.

Example: if ---> to check the condition.

They are 35 keywords in python.

By using this syntax, we can print all the keywords.

```
import keyword
keyword.kwlist
```

['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

```
len(keyword.kwlist)
35
```

Special keywords(True, False, None)
- It is starting with the capital letter(uppercase).
- They can be used as keyword and value.

Practical proof

```
a=10
b=True
```

```
b
True
b=False
b
False
c=None
c
d=and
SyntaxError: invalid syntax
d=as
SyntaxError: invalid syntax
d=assert
SyntaxError: invalid syntax
```

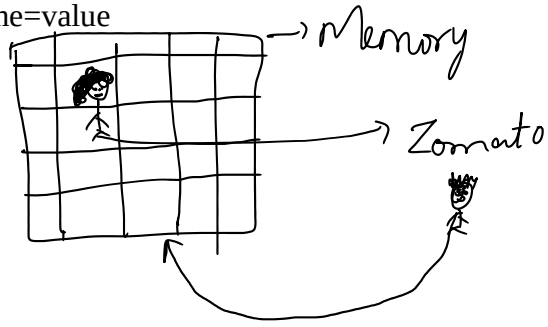
Special keywords got their unique values.
True - 1
False - 0

Variable:
It is a container which is used to store the address of the value stored in the memory.

Or

It is a name given to a container where we store the value.

Syntax: var_name=value

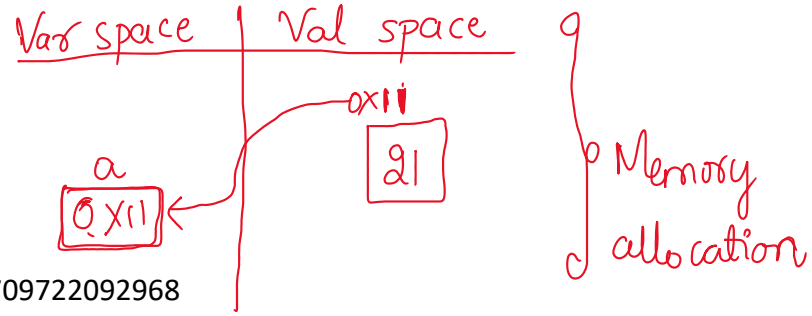



```
a=21
```

id():It is an inbuilt function which is used to get the actual address of the value stored inside the memory.

Syntax: id(var_name/value)

```
a=21
```




```
id(a)
140709722092968
id(21)
140709722092968
```

Multiple Variable Creation: Creating multiple variables in a single line.

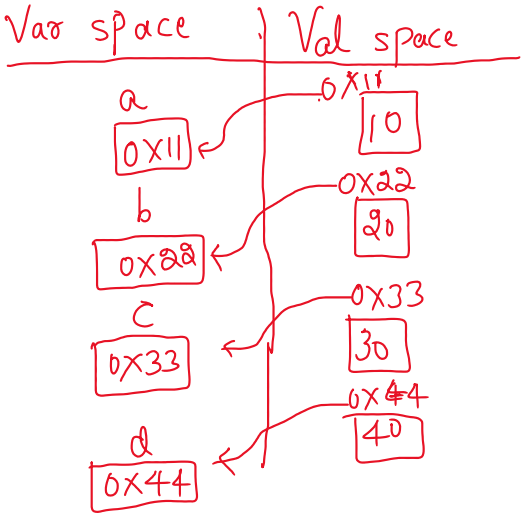
Syntax:
var1,var2,…..,varn=val1,val2,…..,valn

Note:

- Number of variables must be equal to number of values.

a,b,c,d=10,20,30,40

Memory allocation:



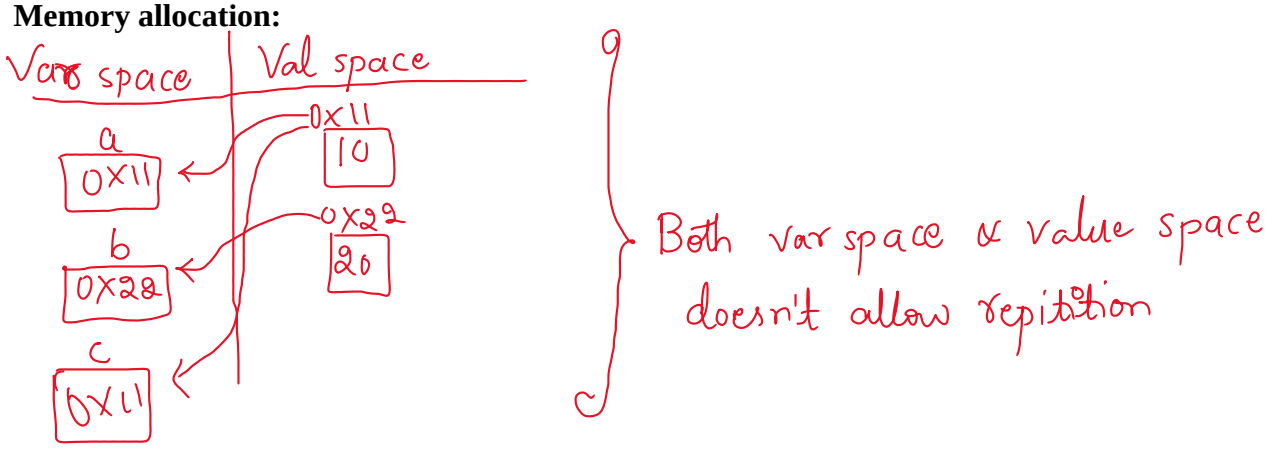

```
a,b,c,d=10,20,30,40
a
10
b
20
c
30
d
40
id(a)
140709722092616
id(b)
140709722092936
id(c)
140709722093256
id(d)
140709722093576
```

What if I don’t follow the note?

```
a,b,c=10,20
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a,b,c=10,20
ValueError: not enough values to unpack (expected 3, got 2)
a,b,c=10,20,30,40
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a,b,c=10,20,30,40
ValueError: too many values to unpack (expected 3)
```

What happens if multiple variables have same value?
a,b,c=10,20,10

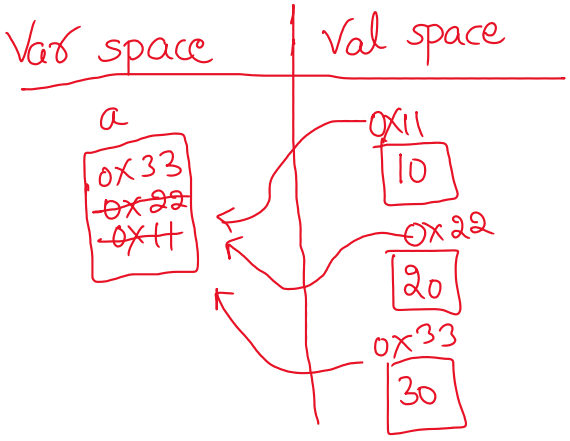
Memory allocation:



What happens if we have same variables ?

a,a,a=10,20,30

Memory allocation:



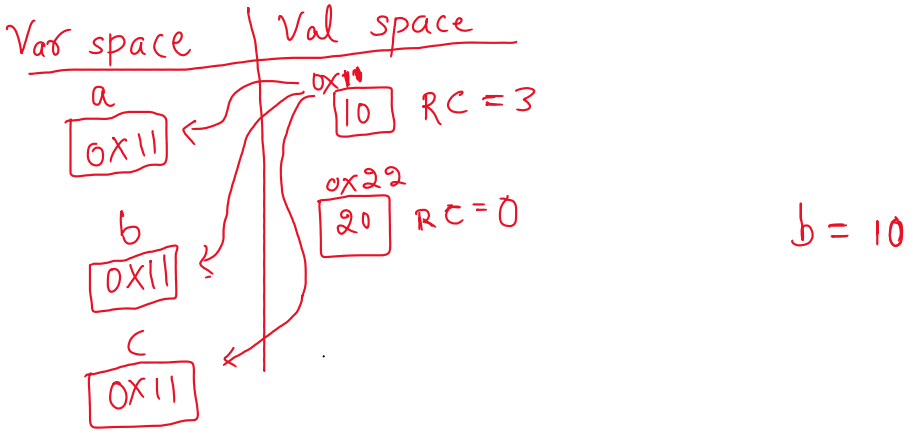
a,b,c=10,20,10
a
10
b
20
c
10
id(a)
140709722092616
id(b)
140709722092936
id(c)
140709722092616

a,a,a=10,20,30
a
30

Reference Count: It will count the number of variables sharing the same value.

a,b,c=10,20,10

Memory allocation:

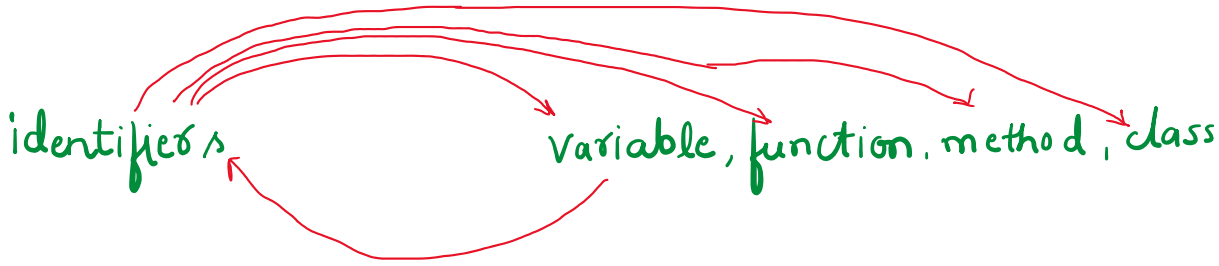


Conclusion: As soon the RC becomes zero(0) , it will be deleted from the memory and it will be collected by garbage collector.

Day-4

Identifiers: It is a name given to a variable uniquely which is used to identify the value.

Note:
All the variables are identifiers, but all the identifiers are not variables.



Rules of Identifiers:

1) Identifiers should not be a keyword.

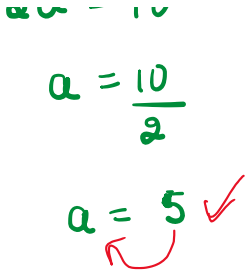
```
a=10
if=10
SyntaxError: invalid syntax
and=10
SyntaxError: invalid syntax
True=10
SyntaxError: cannot assign to True
a=True
a
True
```

Reason: Keywords are already predefined with some task, and special keywords can be used as value but not as variable.

2) Identifier should not start with number.


```
2a=10
```

```
2a=10
SyntaxError: invalid decimal literal
```



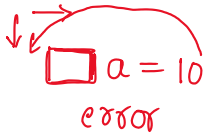
3) Identifier should not contain any special character except underscore(_).

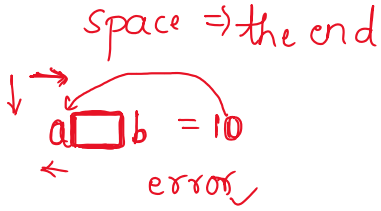
```
a+b=10
SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of '='?
a-b=30
SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of '='?
a_b=40
a_b
40
b_=10
b_
10
_a=10
_a
10
_=20
_
20
```

Reason: All the special characters are predefined with some task, except underscore.

4) Identifier should not contain space in between and at the beginning.

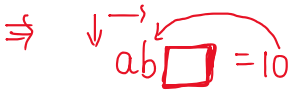
```
a b=10
SyntaxError: invalid syntax

 a=10
SyntaxError: unexpected indent
```



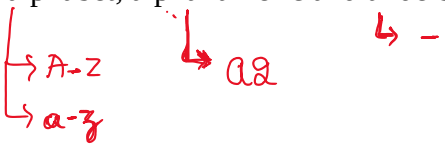


a b

```
ab =10
ab
10
```



5) Identifiers can be alphabet, alphanumeric and underscore.

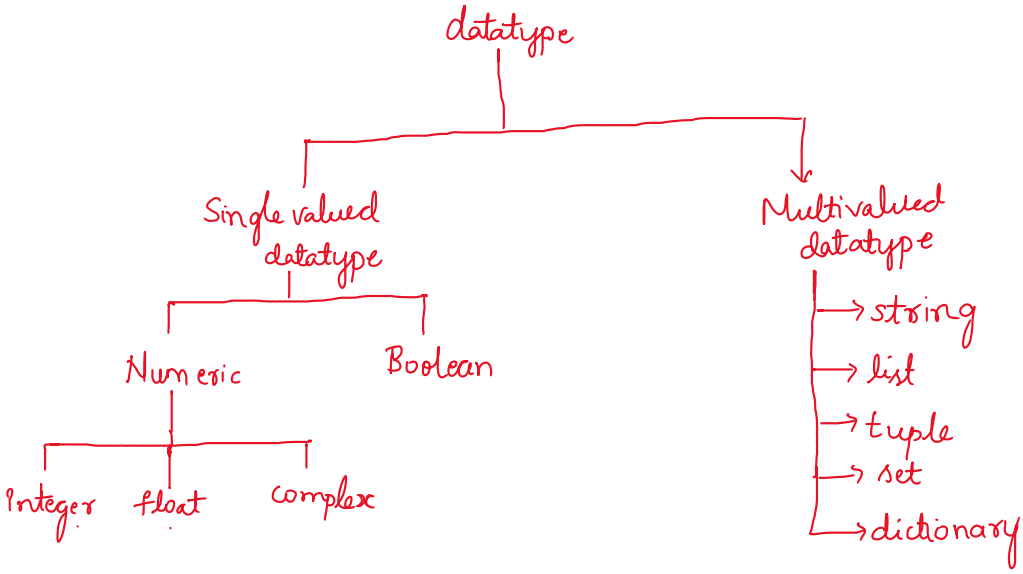


6) Industrial Standard Rule(ISR):
Identifier name should not cross more than 79 characters.

Day-5

Datatypes:
--- It is going to specify the size and type of the value stored in the variable.

Types:



<u>Single valued Datatype</u>

1) Integer:

- It is a real number without decimal point.

In every datatype we are going to common concept called,

- Default value : It is an initial value from where my datatype values starts. These are internally equal to False.
- Non-default value : All the values except Default value are considered as non-default value. These are internally equal to True.

Default value of integer = 0

There are 2 inbuilt function which is common for every datatype.

- type() : Whenever we want to check datatype of the value we use this.

  Syntax: type(var/val)

```
a=10
type(a)
<class 'int'>
type(10)
<class 'int'>
```

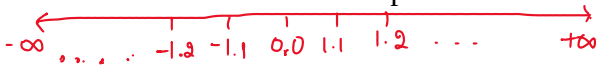
- bool() : Whenever we want to check the value we stored is default value or not.

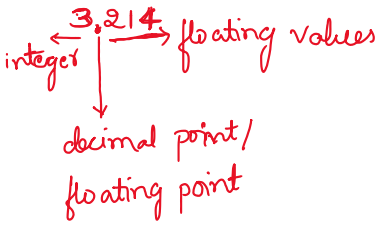
Syntax: bool(var/val)

```
b=0
bool(b)
False
c=12
bool(c)
True
```

2) Float:

- It is a real number with decimal point.





- Default value of float = 0.0
- type():

```
a=1.2
type(a)
<class 'float'>
type(1.2)
<class 'float'>
```

- bool():

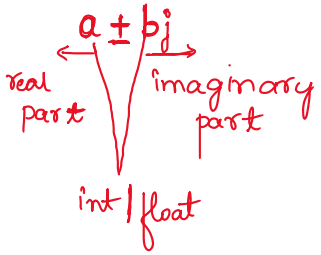
```
b=3.2
bool(b)
True
c=0.0
bool(c)
False
```

- Float datatype values takes max 18 slots.

```
a=1.4543593475934029563966 56
a
1.454359347593403
a=22.3243285742957349644765 6756
a
22.324328574295734
b=2324.3534654765876789798708970
b
2324.3534654765876
```

3) Complex:

- It is a combination of real part and imaginary part.




```
a=2+3j
a
(2+3j)
b=2.3-6.8j
b
(2.3-6.8j)
```

```
c=2+3.5j
c
(2+3.5j)
d=7.1-2j
d
(7.1-2j)
```

- Default value of complex = 0j
- type():

```
a=2+3j
type(a)
<class 'complex'>
type(2+3j)
<class 'complex'>
```

- bool():

```
b=7+3.2j
bool(b)
True
c=0.0+0.0j
bool(c)
False
c
0j
```

Can we use other alphabet instead of j?

```
a=2+3i
SyntaxError: invalid decimal literal
a=6+7b
SyntaxError: invalid decimal literal
b=4+7J
b
(4+7j)
```

Can we interchange the real and imaginary part?

```
a=2x+4j
SyntaxError: invalid decimal literal
b=4j+2
b
(2+4j)
```

But , we cannot interchange the imaginary part values

```
c=3+j8
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    c=3+j8
NameError: name 'j8' is not defined
```

Real time application:

-- In data analytics and large projects.

Day-6

4) Boolean:

It consists of 2 values,

- True
- False

Default value of boolean datatype = False
Non-default value of boolean datatype = True

```
a=True
type(a)
<class 'bool'>
bool(a)
True
b=False
type(b)
<class 'bool'>
bool(b)
False
```

These boolean values can be used in 2 scenarios,

- They can be used as values.

```
a=True
```

- They are used as resultant.

```
100>20
True
10<5
False
```

Multivalued Datatype

5) String:
--- It is a collection of characters enclosed with ' ','' '','' ''
(A-Z,a-z,0-9, special characters).

Syntax: var='val1val2val3……valn'

When we use the three different quotes?

```
a='python'
a
'python'
a="python"
a
'python'
a='''python'''
a
'python'
```

- When we have apostrophe symbol, then we use double quotes.

```
a="ram's age is 19"
a
"ram's age is 19"
```

- When we have paragraph type string we use triple quotes.

```
s='''python is cute
python is easy
```

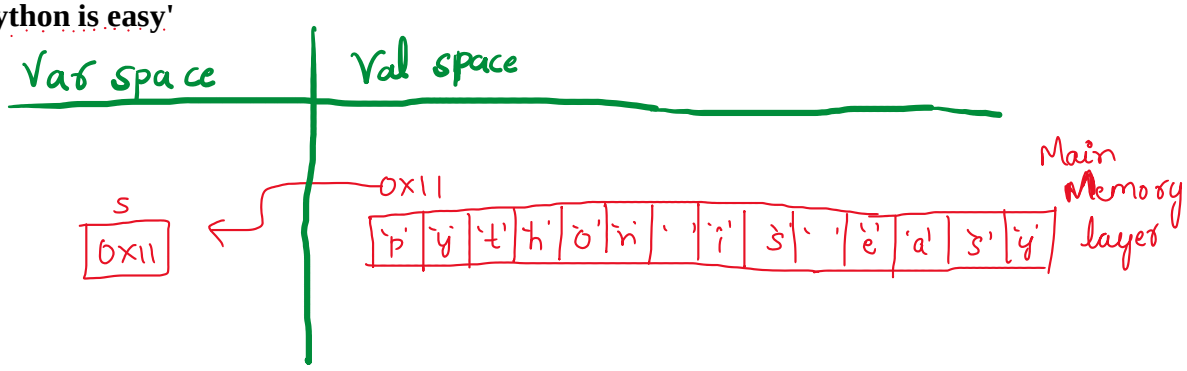
python is king'''

Default value of string = ''

```
a='sakshi'
type(a)
<class 'str'>
bool(a)
True
b=""
bool(b)
False
```

Memory Allocation:

s='python is easy'

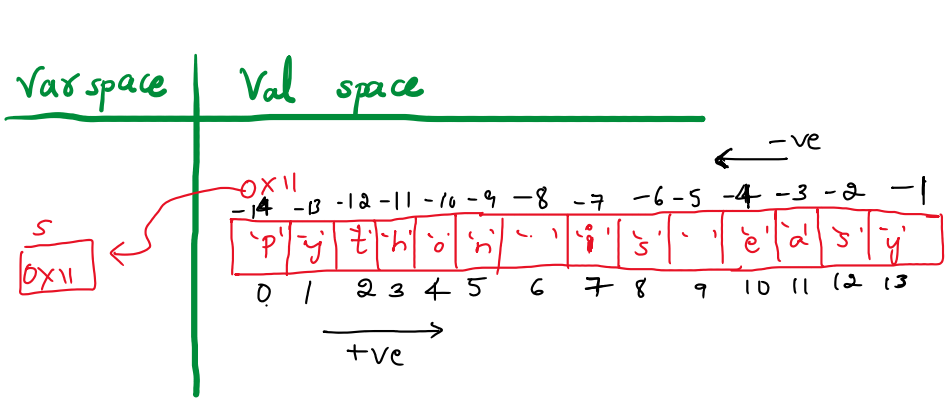


Indexing : The phenomenon of passing subaddress to the memory block is called as Indexing.

Types:

- **+ve indexing --- starts from left to right (starts from 0, ends with len(collection)-1)**
- **-ve indexing --- starts from right to left (starts from -1, ends with -len(collection))**

s='python is easy'



len(s) - 1
=> 14 - 1
=> 13

Note:

- **To get/access the value from the collection, we use the syntax**
  var[index]

```
s='python is easy'
s[4]
'o'
```

```
s[-10]
'o'
```

- **In order to do modification , we use the syntax**
  var[index]=new_value

```
s='python is easy'
s[4]
'o'
s[4]='m'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    s[4]='m'
TypeError: 'str' object does not support item assignment
```

- If the datatype accepts modification, it will be called as **Mutable** datatype.
- If the datatype does not accept modification, it will be called as **Immutable** datatype

Conclusion: String is Immutable datatype.

6) List:

--- It is a collection of homogeneous and heterogeneous values enclosed by [].

Homogeneous --- collection of values of same datatype --- [10,20,30,40]
Heterogeneous --- collection of values of different datatype --- [10,2.0,3+0j,True,'40']

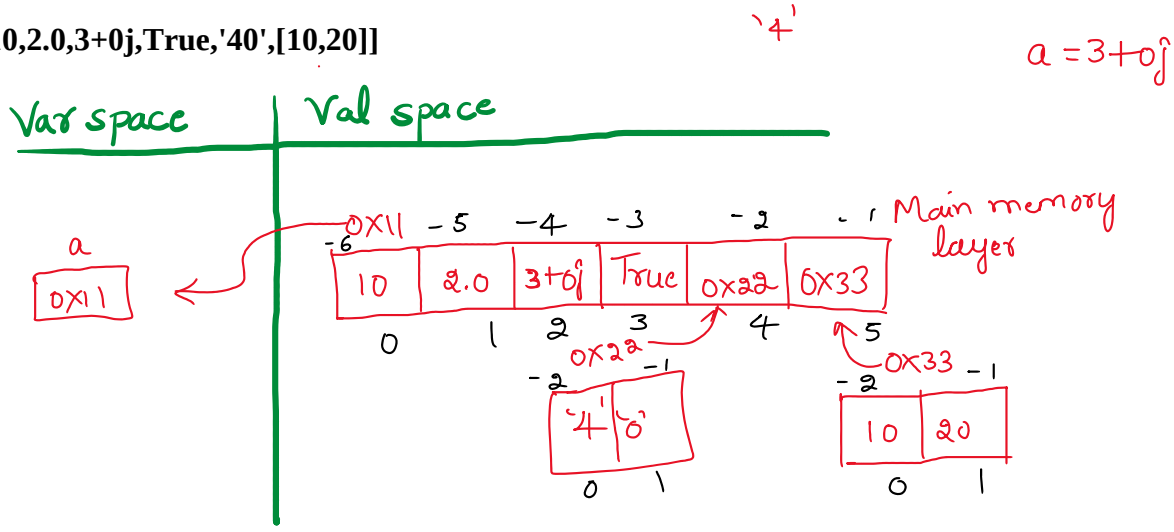
Syntax: var=[val1,val2,val3,…….valn]

Example: l=[10,20,30,40]

Default value of list = []

Memory allocation:

a=[10,2.0,3+0j,True,'40',[10,20]]



var[index]

```
a=[10,2.0,3+0j,True,'40',[10,20]]
a[4]
'40'
a[4][1]
```

```
'0'
a[5]
[10, 20]
a[5][0]
10
a=[10,2.0,3+0j,True,'4',[10,20]]
a[4]
'4'
a[4][0]
'4'
a[-1]
[10, 20]
a[-1][-1]
20
```

var[index]=new_value

```
a=[10,2.0,3+0j,True,'4',[10,20]]
a[5]
[10, 20]
a[5][0]
10
a[5][0]='10'
a
[10, 2.0, (3+0j), True, '4', ['10', 20]]
a[5][1]='20'
a
[10, 2.0, (3+0j), True, '4', ['10', '20']]
```

Conclusion: List is mutable datatype.

```
a=[10,2.0,3+0j,True,'4',[10,20]]
a[4]='sam'
a
[10, 2.0, (3+0j), True, 'sam', [10, 20]]
a[4]
'sam'
a[4][1]
'a'
a[4][1]='m'
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    a[4][1]='m'
TypeError: 'str' object does not support item assignment
```

Day-7

Inbuilt Functions is List:

- **append():**To add a new value to the list.
  Syntax: var.append(val) ----- It will add the value to the last position.

```
a=[10,2.0,3+0j,True,'4',[10,20]]
a.append(7)
a
[10, 2.0, (3+0j), True, '4', [10, 20], 7]
a.append(8)
a
[10, 2.0, (3+0j), True, '4', [10, 20], 7, 8]
a.append('python')
a
[10, 2.0, (3+0j), True, '4', [10, 20], 7, 8, 'python']
```

- **insert():** To add a new value to the list wherever we want.
  Syntax: var.insert(index,val)

```
a=[10,2.0,3+0j,True,'4',[10,20]]
a.insert(4,'Java')
a
[10, 2.0, (3+0j), True, 'Java', '4', [10, 20]]
a.insert(0,2+0j)
a
[(2+0j), 10, 2.0, (3+0j), True, 'Java', '4', [10, 20]]
a.insert(10,'cute')
a
[(2+0j), 10, 2.0, (3+0j), True, 'Java', '4', [10, 20], 'cute']
a.insert(100,'promise day')
a
[(2+0j), 10, 2.0, (3+0j), True, 'Java', '4', [10, 20], 'cute', 'promise day']
a.insert(4,False)
a
[(2+0j), 10, 2.0, (3+0j), False, True, 'Java', '4', [10, 20], 'cute', 'promise day']
a.insert(98,'idiot')
a
[(2+0j), 10, 2.0, (3+0j), False, True, 'Java', '4', [10, 20], 'cute', 'promise day', 'idiot']
```

- **pop():** Used to eliminate value from the list.
  Syntax: var.pop() ----- Last value of the list gets removed.
  var.pop(index) --- removes value from particular index.

```
l=[10, 2.0, (3+0j), True, 'Java', '4', [10, 20]]
l.pop()
[10, 20]
l
[10, 2.0, (3+0j), True, 'Java', '4']
l.pop(3)
True
l
[10, 2.0, (3+0j), 'Java', '4']
l.pop(3)
'Java'
l
[10, 2.0, (3+0j), '4']
```

```
l=[10, 2.0, (3+0j), True, 'Java', '4', [10, 20]]
l.pop([6][0])
[10, 20]
l
[10, 2.0, (3+0j), True, 'Java', '4']
```

- **remove():** Used to remove the value when we don't know the index position of the value.
  syntax: var.remove(value)

```
l=[10, 2.0, (3+0j), True, 'Java', '4', [10, 20]]
l.remove('Java')
l
[10, 2.0, (3+0j), True, '4', [10, 20]]
l.remove(3+0j)
l
[10, 2.0, True, '4', [10, 20]]
```

```
a=[10,2.3,10,2.3,True,True]
a.remove(10)
a
[2.3, 10, 2.3, True, True]
```

```
a.remove(2.3)
a
[10, 2.3, True, True]
```

7) **Tuple :**It is a collection of homogeneous and heterogeneous values enclosed by ().

Syntax:

var=(val1,val2,……..,valn) --- when we have multiple values
var=(val1,) --- when we have only one value.

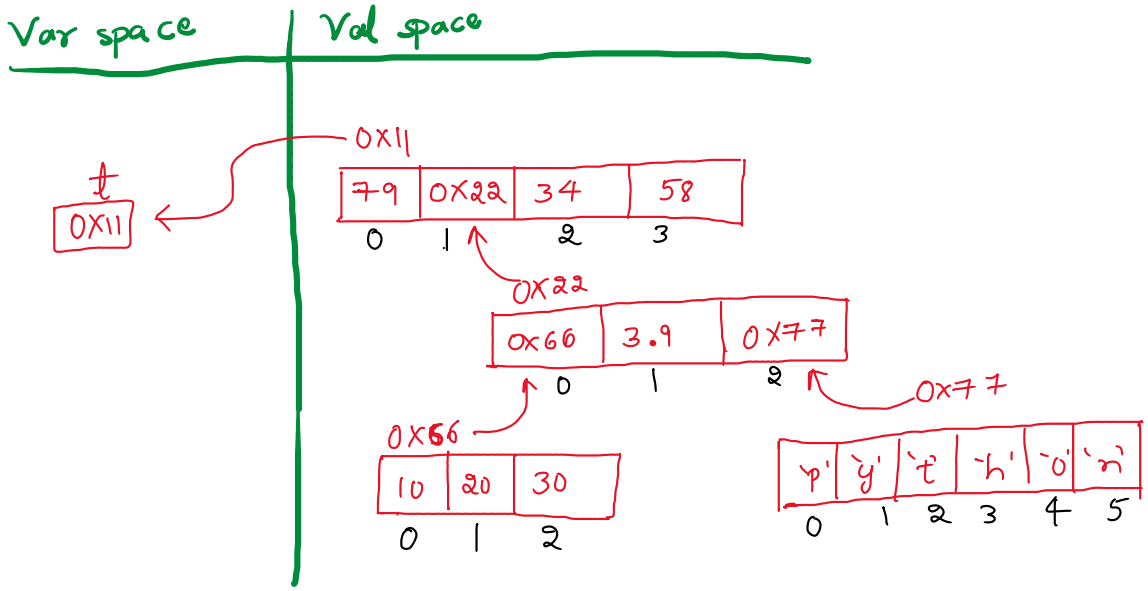
parenthesis is not mandatory

```
a=10,20,30,40
a
(10, 20, 30, 40)
type(a)
<class 'tuple'>
b=(10)
type(b)
<class 'int'>
c=(10,)
type(c)
<class 'tuple'>
d=2.3,
type(d)
<class 'tuple'>
```

Default value of tuple = ()

Memory Allocation:

t=(79,([10,20,30],3.9,'python'),34,58)



- **var[index] --- accessing the value.**

```
t=(79,([10,20,30],3.9,'python'),34,58)
t[1]
([10, 20, 30], 3.9, 'python')
t[0]
79
t[1][0]
[10, 20, 30]
t[2]
34
```

```
t[1][2]
'python'
t[1][0][1]
20
t[1][2][4]
'o'
```

- **var[index]=new_value --- doing modification**

```
t=(79,([10,20,30],3.9,'python'),34,58)
t[1]
([10, 20, 30], 3.9, 'python')
t[1][2]
'python'
t[1][2][2]
't'
t[1][2][2]='@'
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    t[1][2][2]='@'
TypeError: 'str' object does not support item assignment
t[3]
58
t[3]=65
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    t[3]=65
TypeError: 'tuple' object does not support item assignment
t[1][1]
3.9
t[1][0]
[10, 20, 30]
t[1][0][1]
20
t[1][0][1]=50
t
(79, ([10, 50, 30], 3.9, 'python'), 34, 58)
t[1][0]
[10, 50, 30]
t[1][0]=[1,2]
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    t[1][0]=[1,2]
TypeError: 'tuple' object does not support item assignment
t[1][1]
3.9
t[1][1]=8.9
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    t[1][1]=8.9
TypeError: 'tuple' object does not support item assignment
```

Conclusion: Tuple is immutable datatype.

Note:	**Tuple is the most secured datatype of python.**

Day-8

Set:

--- It is a collection of homogeneous and heterogeneous values enclosed by {}.

syntax:

var={val1,val2,……,valn}

Default value of set = set()

Important points for set datatype:

- Values should be of immutable type(int, float, complex, bool, str, tuple).

```
a={10,2.3,2+3j,True,'hi',(10,20)}
a={10,2.3,2+3j,True,'hi',(10,20),[30,40]}
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a={10,2.3,2+3j,True,'hi',(10,20),[30,40]}
TypeError: unhashable type: 'list'
```

unhashable ⇒ mutable
hashable ⇒ immutable

- Set is unordered in nature.

```
a={10,2.3,2+3j,True,'hi',(10,20)}
A
{True, 2.3, 'hi', 10, (2+3j), (10, 20)}
```

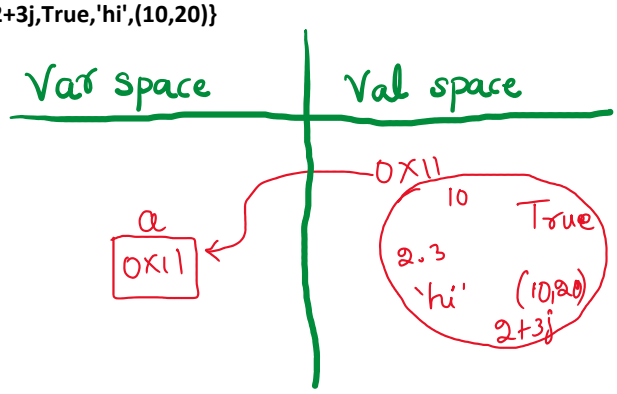
Unordered --- order of input does not match the order of output

- Set will eliminate repeated/duplicate values.

```
a={10,2.3,2+3j,10,True,2.3,'hi',(10,20),1}
a
{True, 2.3, 10, (2+3j), 'hi', (10, 20)}
```

Memory allocation:

a={10,2.3,2+3j,True,'hi',(10,20)}



⇒ all the values are stored in the hash tables.

⇒ hash values are required to store them into hash values

```
hash(10)
10
hash(2.3)
691752902764107778
hash(2+3j)
3000011
hash(True)
1
hash('hello')
-6453495032443784337
hash((10,20))
-4873088377451060145
hash([10,20])
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
```

```
  hash([10,20])
TypeError: unhashable type: 'list'
```

- Set will not support for indexing.

- Inbuilt functions to do modification for set:
  - add() --- To add a value to a set.
    Syntax:
        var.add(val)

```
a={10,2.3,2+3j,True,'hi',(10,20)}
a.add('python')
a
{True, 2.3, 'hi', 'python', 10, (2+3j), (10, 20)}
a.add('hi')
a
{True, 2.3, 'hi', 'python', 10, (2+3j), (10, 20)}
```

  - remove() --- To remove the value from the set.
    Syntax:
        var.remove(val)

```
a={10,2.3,2+3j,True,'hi',(10,20)}
a.remove(2+3j)
a
{True, 2.3, 'hi', 10, (10, 20)}
a.remove(10)
a
{True, 2.3, 'hi', (10, 20)}
a.remove(2.3)
a
{True, 'hi', (10, 20)}
```

  - pop() --- To remove very first value in the set.
    Syntax:
        var.pop()

```
s={True, 'hi', (10, 20)}
s.pop()
True
s
{'hi', (10, 20)}
s.pop()
'hi'
s
{(10, 20)}
s.pop()
(10, 20)
s
set()
```

Conclusion: Set is mutable datatype.

9) Dictionary:

--- It is a collection of key-value pairs enclosed by {} and the key-value pair is seperated by colon(:).

Syntax:
    var={k1:v1,k2:v2,……..,kn:vn}

Default value of dictionary = {}

Note points for dictionary:

- Keys should be immutable, but value can be of any datatype.

```
a={'a':10,[10]:20,{10,20}:30}
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    a={'a':10,[10]:20,{10,20}:30}
TypeError: unhashable type: 'list'
a={'a':10,10:20,{10,20}:30}
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    a={'a':10,10:20,{10,20}:30}
TypeError: unhashable type: 'set'
```

- Keys should be unique. If we try to enter the same key then old value of that key will be replaced by new value.

```
a={'a':10,'b':20,'a':30}
a
{'a': 30, 'b': 20}

a={'a':10,'b':10}
a
{'a': 10, 'b': 10}
```

- Dictionary will not support for indexing.

```
a={'a':10,'b':20,'c':30}
a[1]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a[1]
KeyError: 1
```

Note:
--- In dictionary, only the keys are the visible layer.

- To access the value of dictionary we have to use the syntax,
  var[key]

```
s={'a':10,'b':20,'c':30}
s['b']
20
s['a']
10
s['c']
30
```

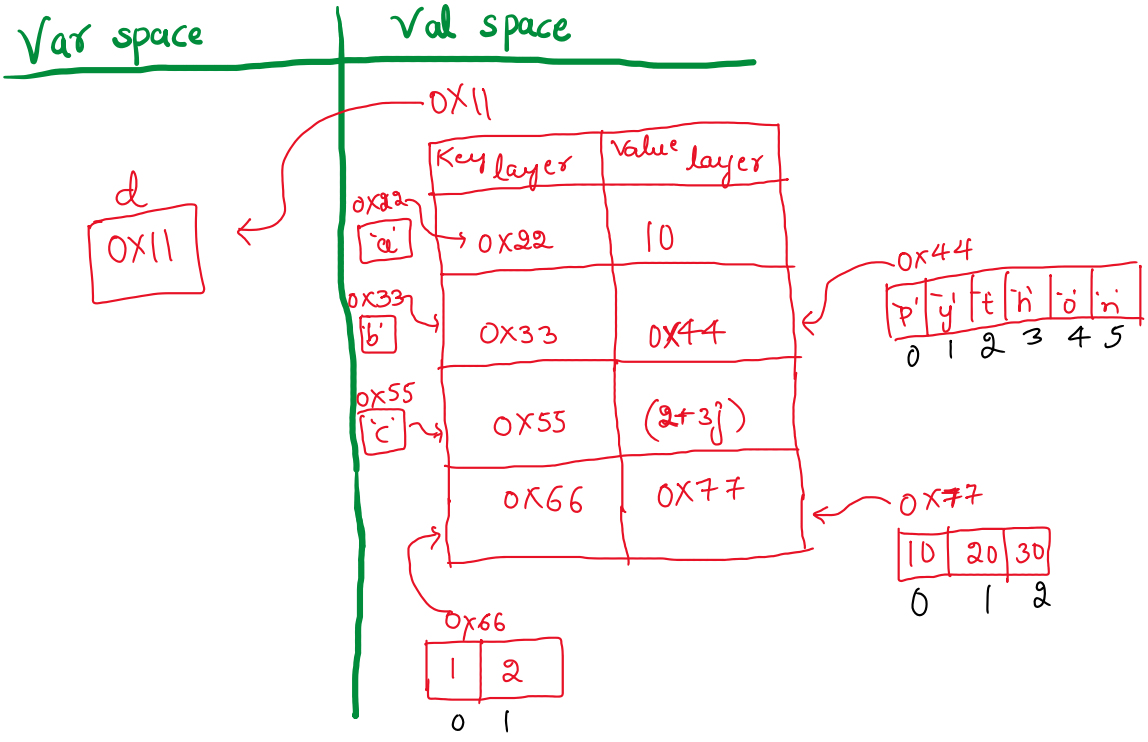
- To do modification we have to use the syntax,
  var[key]=new_value

```
s={'a':10,'b':20,'c':30}
s['c']=40
s
{'a': 10, 'b': 20, 'c': 40}
s['b']=100
s
{'a': 10, 'b': 100, 'c': 40}
```

Conclusion: Dictionary is mutable datatype.

Memory allocation:

d={'a':10,'b':'python','c':2+3j,(1,2):[10,20,30]}




```
d={'a':10,'b':'python','c':2+3j,(1,2):[10,20,30]}
d['b']
'python'
d['b'][3]
'h'
d[(1,2)]
[10, 20, 30]
d[(1,2)][2]
30
```

Day-9

Slicing:
--- It is used to extract the group of values from the collection.

Syntax:
    var[SI:EI+1:updation]
    var[SI:EI-1:updation]

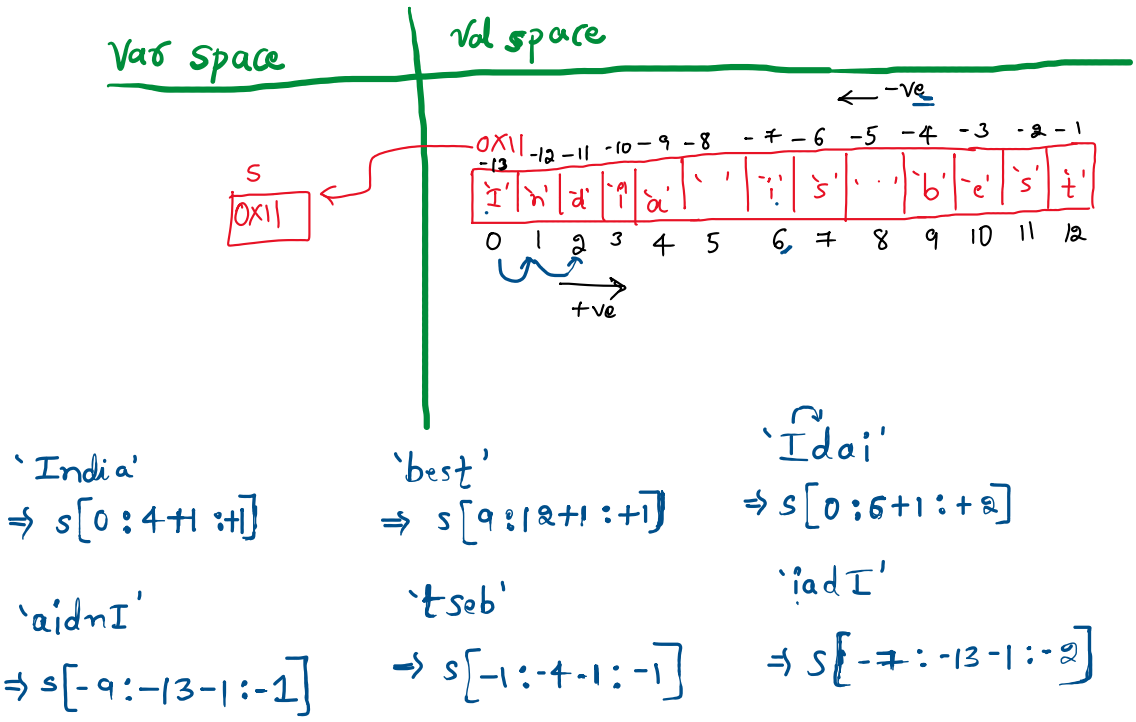
Slicing supports on String, List and Tuple.

Note:

- When we extract values from left to right
  EI+1
- When we extract values from right to left
  EI-1

Memory allocation:

s='India is best'




```
s='India is best'
s[0:5:1]
'India'
s[0:4+1:+1]
'India'
s[9:13:1]
'best'
s[0:7:2]
'Idai'
s[-9:-14:-1]
'aidnI'
s[-1:-5:-1]
'tseb'
s[-7:-14:-2]
'iadI'
```

Slicing shortcuts:

- **If SI is 0 or -1**

  [:EI+1:updation]
  [:EI-1:updation]

- **If EI is the end of the collection.**

  [SI::updation]
  [SI::updation]

- **If Updation is +1**

  [SI:EI:]

```
s='India is best'
s[::]
'India is best'
s[::-1]
```

'tseb si aidnl'

Typecasting / Type Conversion :

--- The phenomenon of converting the data/value from one datatype to another.

Syntax: dest_type(source_var)

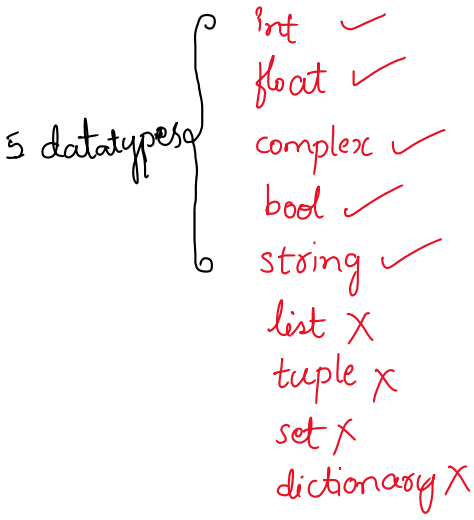
Note:

- **Converting from SVDT to MVDT only string is supported**
- **Converting from MVDT to SVDT only boolean is supported**

1) Converting from int to other datatypes:
a=10

For SVDT:

```
a=10
type(a)
<class 'int'>
int(a)
10
float(a)
10.0
complex(a)
(10+0j)
complex(a,a)
(10+10j)
bool(a)
True
```

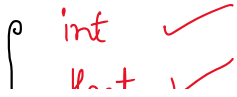


For MVDT:

```
str(a)
'10'
list(a)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    list(a)
TypeError: 'int' object is not iterable
tuple(a)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    tuple(a)
TypeError: 'int' object is not iterable
set(a)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    set(a)
TypeError: 'int' object is not iterable
dict(a)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    dict(a)
TypeError: 'int' object is not iterable
```

1) Converting from float to other datatypes:
a=7.4

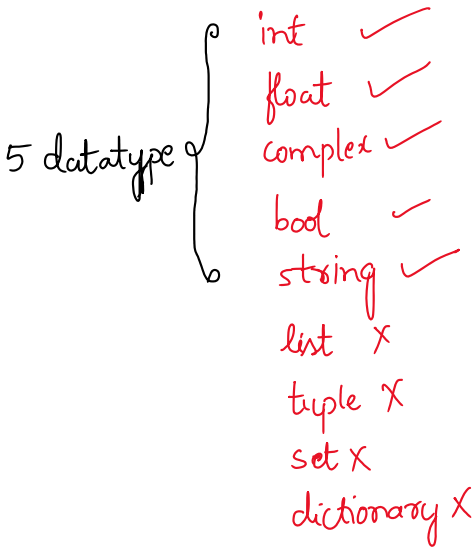
For SVDT:



a=7.4

For SVDT:

```
a=7.4
type(a)
<class 'float'>
int(a)
7
complex(a)
(7.4+0j)
complex(a,a)
(7.4+7.4j)
bool(a)
True
```



For MVDT:

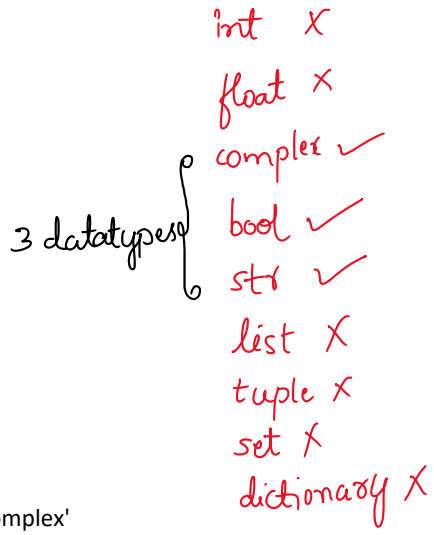
```
str(a)
'7.4'
list(a)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    list(a)
TypeError: 'float' object is not iterable
tuple(a)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    tuple(a)
TypeError: 'float' object is not iterable
set(a)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    set(a)
TypeError: 'float' object is not iterable
dict(a)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    dict(a)
TypeError: 'float' object is not iterable
```

3) Converting from complex to other datatypes:

a=7+3j

For SVDT:

```
a=7+3j
type(a)
<class 'complex'>
int(a)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int(a)
TypeError: int() argument must be a string, a bytes-like
object or a real number, not 'complex'
float(a)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    float(a)
TypeError: float() argument must be a string or a real number, not 'complex'
bool(a)
True
```



For MVDT:

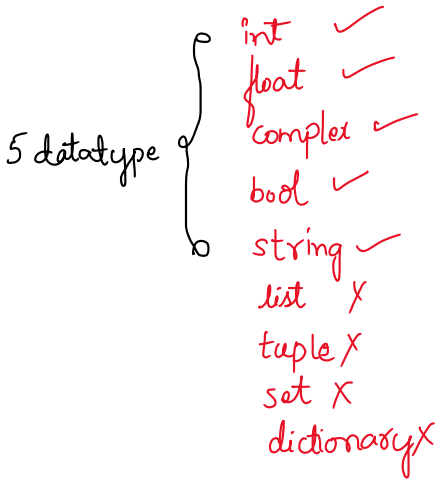
```
str(a)
'(7+3j)'
list(a)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    list(a)
TypeError: 'complex' object is not iterable
tuple(a)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    tuple(a)
TypeError: 'complex' object is not iterable
set(a)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    set(a)
TypeError: 'complex' object is not iterable
dict(a)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    dict(a)
TypeError: 'complex' object is not iterable
```

4) Converting from bool to other datatypes:

a=True

For SVDT:

```
a=True
int(a)
1
float(a)
1.0
complex(a)
(1+0j)
bool(a)
True
complex(a,a)
(1+1j)
```

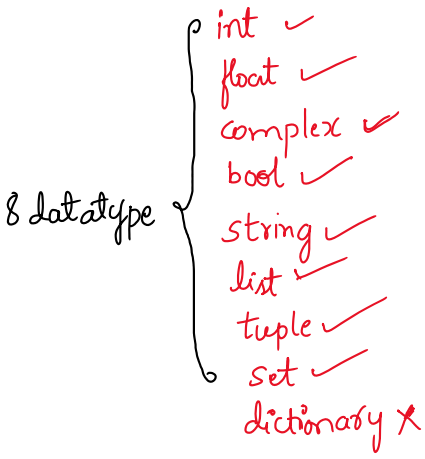


For MVDT:

```
str(a)
'True'
list(a)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    list(a)
TypeError: 'bool' object is not iterable
tuple(a)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    tuple(a)
TypeError: 'bool' object is not iterable
set(a)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    set(a)
TypeError: 'bool' object is not iterable
dict(a)
Traceback (most recent call last):
```

```
  File "<pyshell#11>", line 1, in <module>
    dict(a)
TypeError: 'bool' object is not iterable
```

5) Converting from string to other datatypes: a='python'



For SVDT:

```
a='python'
int(a)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int(a)
ValueError: invalid literal for int() with base 10: 'python'
b='12'
int(b)
12
float(b)
12.0
c='abc123'
int(c)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    int(c)
ValueError: invalid literal for int() with base 10: 'abc123'
complex(b)
(12+0j)
complex(b,b)
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    complex(b,b)
TypeError: complex() can't take second arg if first is a string
complex(12,b)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    complex(12,b)
TypeError: complex() second arg can't be a string
complex(b,10)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    complex(b,10)
TypeError: complex() can't take second arg if first is a string
bool(b)
True
```

For MVDT:

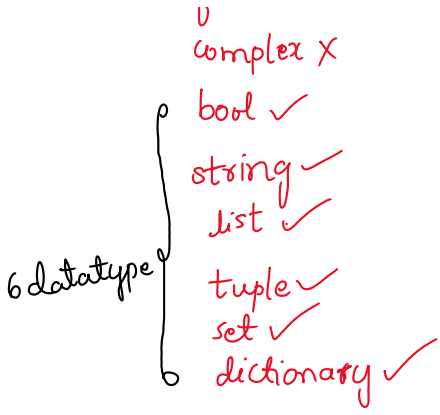
```
a='python'
list(a)
['p', 'y', 't', 'h', 'o', 'n']
tuple(a)
('p', 'y', 't', 'h', 'o', 'n')
set(a)
{'o', 'n', 'p', 't', 'y', 'h'}
dict(a)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    dict(a)
ValueError: dictionary update sequence element #0 has length 1; 2 is required
```

6) Converting from list to other datatypes: a=[10,20,30,40]

For SVDT:

```
a=[10,20,30,40]
type(a)
```

```
a=[10,20,30,40]
type(a)
<class 'list'>
int(a)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int(a)
TypeError: int() argument must be a string, a bytes-like object or a real number, not 'list'
float(a)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    float(a)
TypeError: float() argument must be a string or a real number, not 'list'
complex(a)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    complex(a)
TypeError: complex() first argument must be a string or a number, not 'list'
bool(a)
True
```



For MVDT:

```
str(a)
'[10, 20, 30, 40]'
tuple(a)
(10, 20, 30, 40)
set(a)
{40, 10, 20, 30}
dict(a)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    dict(a)
TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

Note(From list to dict conversion):

- Values inside the list should be collection
- The length of the each and every value should be 2

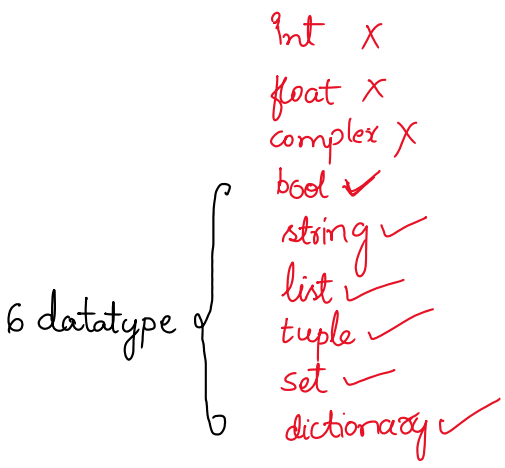
```
b=['hi',[10,20],(30,40),{50,60}]
dict(b)
{'h': 'i', 10: 20, 30: 40, 50: 60}
```

Day-11

7) Converting from tuple to other datatypes:
a=(10,20,30,40)

For SVDT:

```
a=(10,20,30,40)
type(a)
<class 'tuple'>
int(a)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int(a)
TypeError: int() argument must be a string, a bytes-like
object or a real number, not 'tuple'
float(a)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    float(a)
TypeError: float() argument must be a string or a real number, not 'tuple'
```



```
complex(a)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    complex(a)
TypeError: complex() first argument must be a string or a number, not 'tuple'
bool(a)
True
```

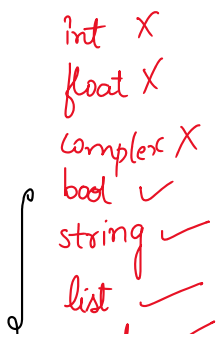
For MVDT:

```
a=(10,20,30,40)
type(a)
<class 'tuple'>
int(a)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int(a)
TypeError: int() argument must be a string, a bytes-like object or a real number, not 'tuple'
float(a)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    float(a)
TypeError: float() argument must be a string or a real number, not 'tuple'
complex(a)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    complex(a)
TypeError: complex() first argument must be a string or a number, not 'tuple'
bool(a)
True
str(a)
'(10, 20, 30, 40)'
list(a)
[10, 20, 30, 40]
set(a)
{40, 10, 20, 30}
dict(a)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    dict(a)
TypeError: cannot convert dictionary update sequence element #0 to a sequence
b=('sa',[10,20])
dict(b)
{'s': 'a', 10: 20}
b=('sa',[10,20,30])
dict(b)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    dict(b)
ValueError: dictionary update sequence element #1 has length 3; 2 is required
```

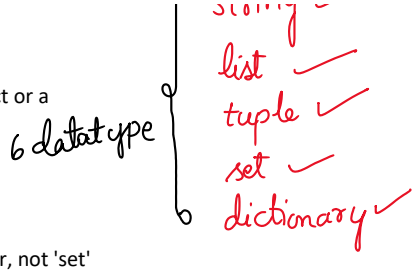
8) Converting from set to other datatypes:
a={10,20,30,40}

For SVDT:

```
a={10,20,30,40}
type(a)
<class 'set'>
int(a)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int(a)
TypeError: int() argument must be a string, a bytes-like object or a
```



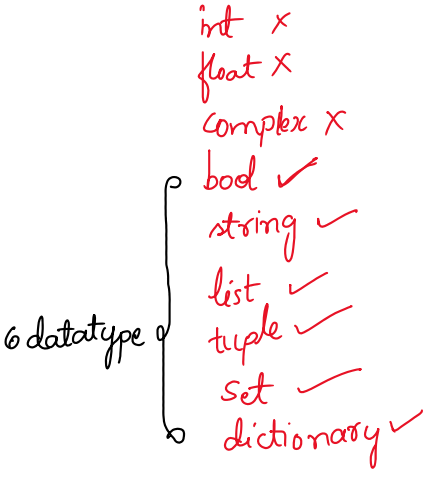
```
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int(a)
TypeError: int() argument must be a string, a bytes-like object or a
real number, not 'set'
float(a)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    float(a)
TypeError: float() argument must be a string or a real number, not 'set'
complex(a)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    complex(a)
TypeError: complex() first argument must be a string or a number, not 'set'
bool(a)
True
```



For MVDT:

```
str(a)
'{40, 10, 20, 30}'
list(a)
[40, 10, 20, 30]
tuple(a)
(40, 10, 20, 30)
dict(a)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    dict(a)
TypeError: cannot convert dictionary update sequence element #0 to a sequence
b={'cd',(10,20)}
dict(b)
{'c': 'd', 10: 20}
```

9) Converting from dictionary to other datatypes: a={'a':10,'b':20,'c':30}



For SVDT:

```
a={'a':10,'b':20,'c':30}
type(a)
<class 'dict'>
int(a)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    int(a)
TypeError: int() argument must be a string, a bytes-like
object or a real number, not 'dict'
float(a)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    float(a)
TypeError: float() argument must be a string or a real number, not 'dict'
complex(a)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    complex(a)
TypeError: complex() first argument must be a string or a number, not 'dict'
bool(a)
True
```

For MVDT:

```
str(a)
```

```
"{'a': 10, 'b': 20, 'c': 30}"
list(a)
['a', 'b', 'c']
list(a.values())
[10, 20, 30]
list(a.items())
[('a', 10), ('b', 20), ('c', 30)]
tuple(a)
('a', 'b', 'c')
tuple(a.values())
(10, 20, 30)
tuple(a.items())
(('a', 10), ('b', 20), ('c', 30))
set(a)
{'a', 'c', 'b'}
set(a.values())
{10, 20, 30}
set(a.items())
{('b', 20), ('c', 30), ('a', 10)}
dict(a)
{'a': 10, 'b': 20, 'c': 30}
```

Copy operation:
--- It is a phenomenon of copying the content from one variable to another variable.

Types:

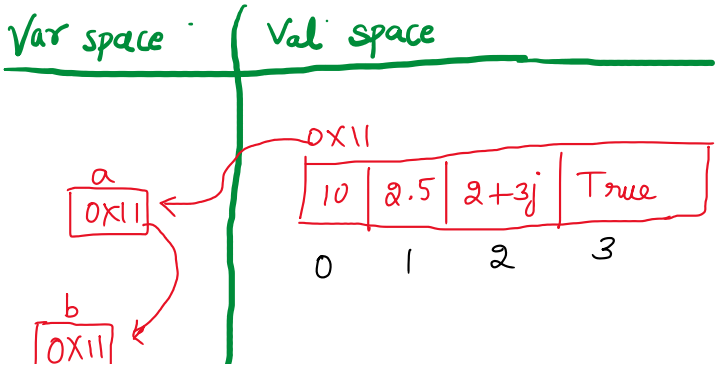
- **General copy/ Normal copy**
- **Shallow copy**
- **Deep copy**

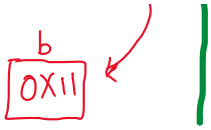
int X
float X
complex X
bool X
String X
list ✓
tuple X
set X
dictionary X

1) General copy:
--- It will just copy the variable space of one variable to another variable.

Syntax: dest_var=source_var

Memory allocation:
Example 1:
a=[10,2.5,2+3j,True] ---- linear collection
b=a

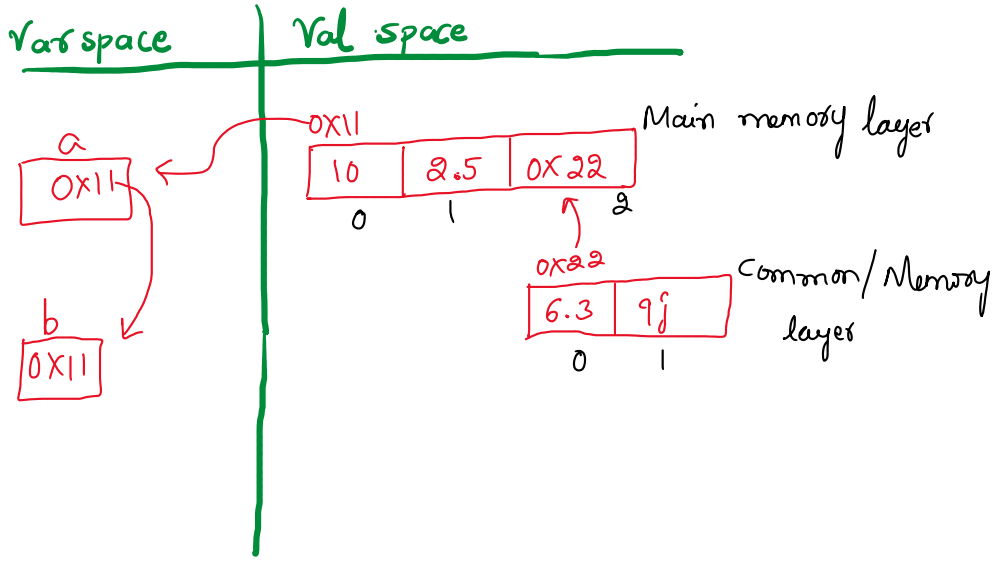





```
a=[10,2.5,2+3j,True]
b=a
a[1]=10
a
[10, 10, (2+3j), True]
b
[10, 10, (2+3j), True]
id(a)
1575699907328
id(b)
1575699907328
```

Example 2:

a=[10,2.5,[6.3,9j]] ----- Nested collection
b=a




```
a=[10,2.5,[6.3,9j]]
b=a
a
[10, 2.5, [6.3, 9j]]
b
[10, 2.5, [6.3, 9j]]
a[1]=[6.3,9j]
a
[10, [6.3, 9j], [6.3, 9j]]
b
[10, [6.3, 9j], [6.3, 9j]]
id(a)
2602509778368
id(b)
2602509778368
```

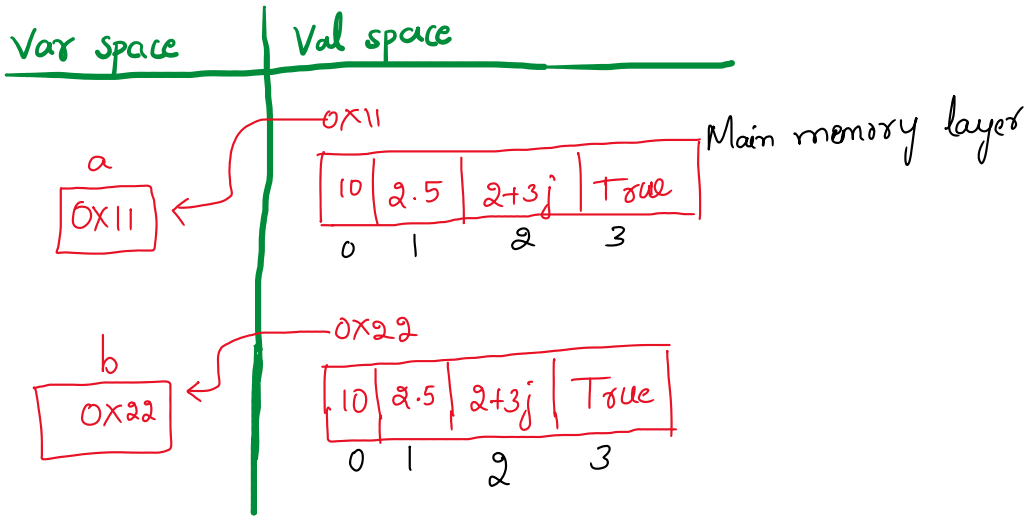
Conclusion: Modification done in linear and nested collection of source variable is affecting the destination variable

2) Shallow copy:

--- It is a phenomenon of copying the content of main memory layer of value space from one variable to another.

Syntax: dest_var=source_var.copy()

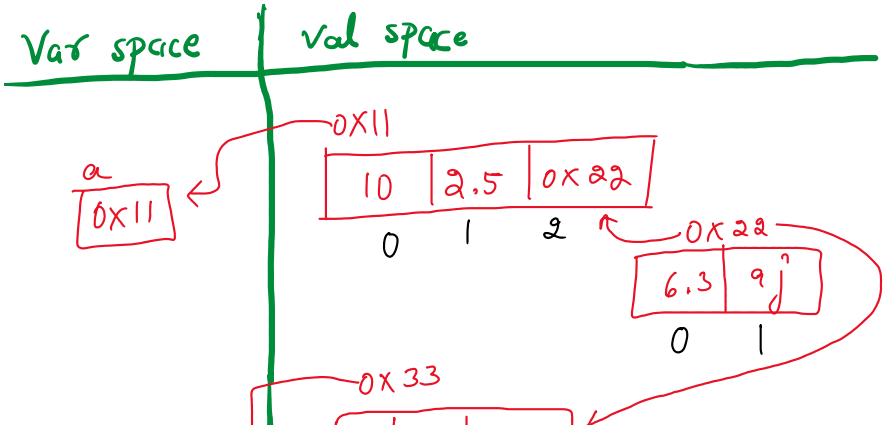
Memory allocation:
Example 1:
a=[10,2.5,2+3j,True] ---- linear collection
b=a.copy()

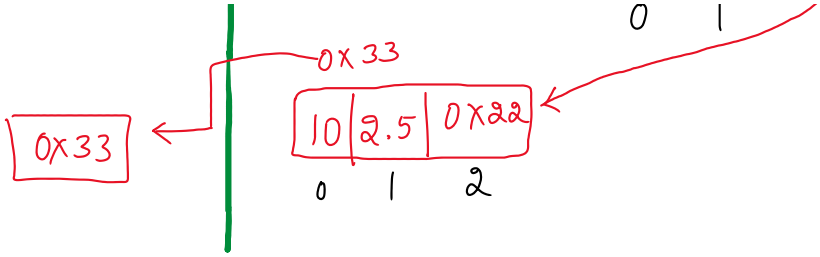



```
a=[10,2.5,2+3j,True]
b=a.copy()
a[2]=4
a
[10, 2.5, 4, True]
b
[10, 2.5, (2+3j), True]
id(a)
1867641833088
id(b)
1867641964160
```

Example 2:

a=[10,2.5,[6.3,9j]] ----- Nested collection
b=a.copy()






```
a=[10,2.5,[6.3,9j]]
b=a.copy()
a
[10, 2.5, [6.3, 9j]]
b
[10, 2.5, [6.3, 9j]]
a[2][0]=True
a
[10, 2.5, [True, 9j]]
b
[10, 2.5, [True, 9j]]
id(a)
2793763676032
id(b)
2793774040512
id(a[2])
2793773909632
id(b[2])
2793773909632
```

Conclusion: Modification done w.r.t linear collection of source variable will not affect the destination variable, but Modification done w.r.t nested collection of source variable will affect the destination variable.

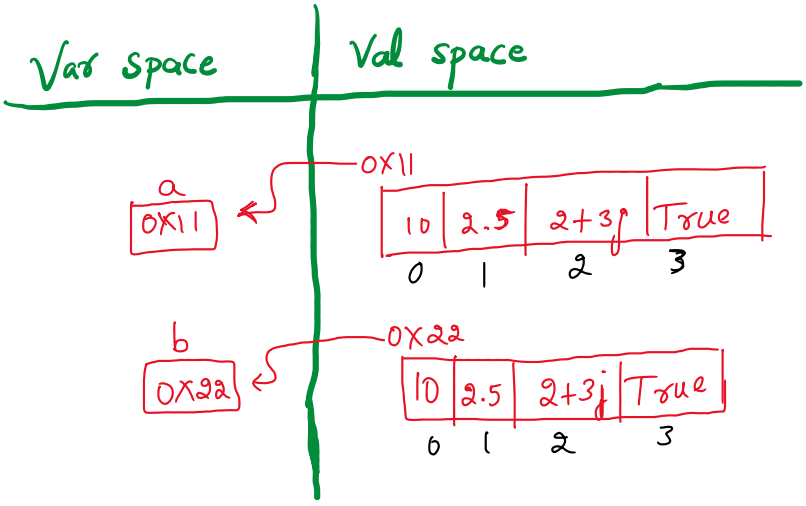
3) Deep copy:

--- It is a phenomenon of copying the entire content of value space from one variable to another.

Syntax:

```
import copy
dest_var=copy.deepcopy(source_var)
```

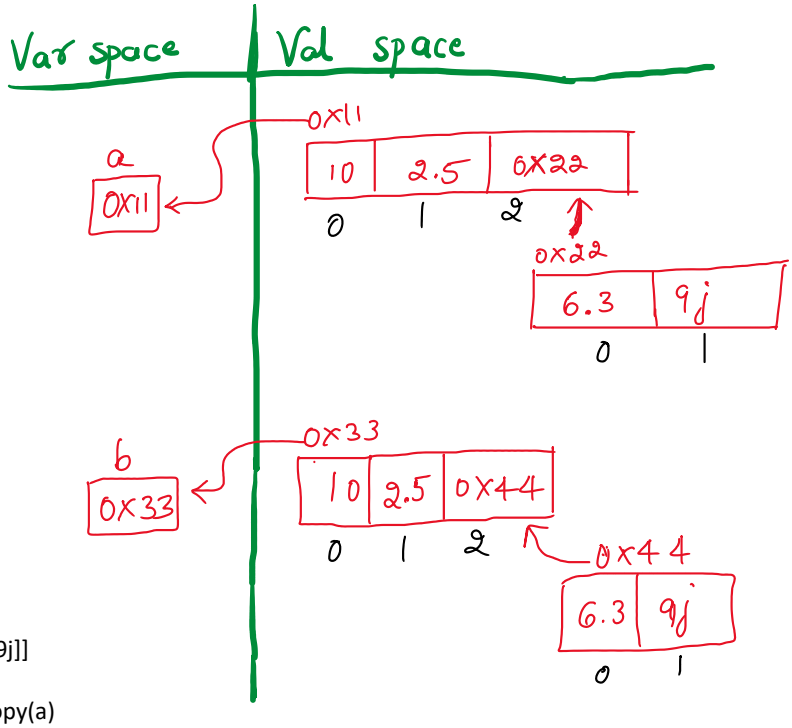
Memory allocation:
Example 1:
a=[10,2.5,2+3j,True] ---- linear collection
import copy
b=copy.deepcopy(a)



```
a=[10,2.5,2+3j,True]
import copy
b=copy.deepcopy(a)
a
[10, 2.5, (2+3j), True]
b
[10, 2.5, (2+3j), True]
a[3]=False
a
[10, 2.5, (2+3j), False]
b
[10, 2.5, (2+3j), True]
id(a)
1746537137792
id(b)
1746537070720
```

Example 2:

a=[10,2.5,[6.3,9j]] ----- Nested collection
import copy
b=copy.deepcopy(a)



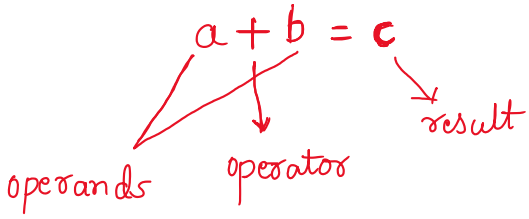

```
a=[10,2.5,[6.3,9j]]
import copy
b=copy.deepcopy(a)
a
[10, 2.5, [6.3, 9j]]
b
[10, 2.5, [6.3, 9j]]
a[2][1]=(2+3j)
a
[10, 2.5, [6.3, (2+3j)]]
b
[10, 2.5, [6.3, 9j]]
id(a)
2516083974016
id(b)
2516094139840
id(a[2])
2516094404224
id(b[2])
```

Conclusion: Modification done in linear and nested collection of source variable will not affect the destination variable.

Note: Deep copy is the only type which satisfies the objective of Copy operation.

Day-12

Operators: They are the special symbols, which is used to perform specific task with the help of operands to give the result.

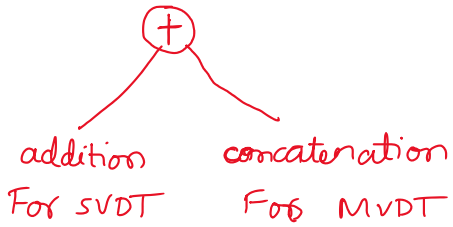


Types:

- **Arithmetic**
- **Logical**
- **Relational/ Comparision**
- **Bitwise**
- **Assignment**
- **Membership**
- **Identity**

1) Arithmetic Operator:

- **Addition(+):**



Syntax: OP1 + OP2

For SVDT:
Note: It will consider the value only

```
10+20
30
2.4+6.8
9.2
(2+3j)+(4+7j)
(6+10j)
True+False
1
1+2.3+(5+3j)+True
(9.3+3j)
```

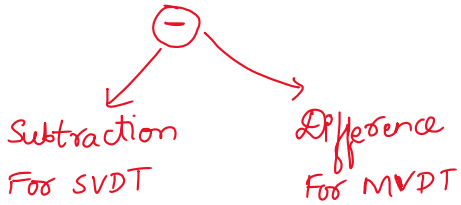
For MVDT:
Note: It will consider both the value and datatype.

```
'hi'+10
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
```

```
  'hi'+10
TypeError: can only concatenate str (not "int") to str
'hi'+'bye'
'hibye'
[10,20,30]+[40,50]
[10, 20, 30, 40, 50]
(10,20,30)+(40,50)
(10, 20, 30, 40, 50)
{10,20,30}+{40,50}
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    {10,20,30}+{40,50}
TypeError: unsupported operand type(s) for +: 'set' and 'set'
{'a':10}+{'b':20}
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    {'a':10}+{'b':20}
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

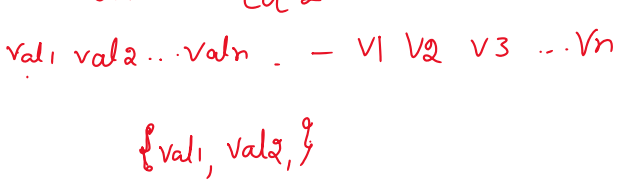
- **Subtraction(-):**

  Syntax: OP1 - OP2



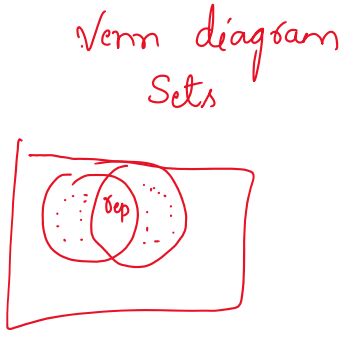
  For SVDT:

```
10-7
3
7.8-5.7
2.0999999999999996
(7+8j)-(2+3j)
(5+5j)
True-True
0
26-6.8-(6+2j)-True
(12.2-2j)
```



  For MVDT:

```
'sam'-'hello'
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    'sam'-'hello'
TypeError: unsupported operand type(s) for -: 'str' and 'str'
[10,20,30]-[20,30]
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    [10,20,30]-[20,30]
TypeError: unsupported operand type(s) for -: 'list' and 'list'
(10,20,30)-(20,30)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    (10,20,30)-(20,30)
TypeError: unsupported operand type(s) for -: 'tuple' and 'tuple'
{10,20,30}-{20,30}
{10}
{'a':10}-{'b':20}
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    {'a':10}-{'b':20}
TypeError: unsupported operand type(s) for -: 'dict' and 'dict'
```



- **Multiplication(*):**

  Syntax: OP1 * OP2

  For SVDT:

```
10*5
50
2.3*7.9
18.169999999999998
(2+3j)*(7+9j)
(-13+39j)
True*False
0
10*2.3*(4+7j)*True
(92+161j)
```

  For MVDT:

```
'python'*'python'
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    'python'*'python'
TypeError: can't multiply sequence by non-int of type 'str'
'python'*4
'pythonpythonpythonpython'
[10,20,30]*2
[10, 20, 30, 10, 20, 30]
(10,20,30)*3
(10, 20, 30, 10, 20, 30, 10, 20, 30)
{10,20,30}*3
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    {10,20,30}*3
TypeError: unsupported operand type(s) for *: 'set' and 'int'
{'a':10}*2
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    {'a':10}*2
TypeError: unsupported operand type(s) for *: 'dict' and 'int'
[10,20]*[20,30]
```

- **Division(): ---- It will support only for SVDT**
  - **True Division(/):** It is used to get the quotient as the output. And the output will be displayed in the float format.

    Syntax: OP1 / OP2

```
10/2
5.0
2.3/1.8
1.2777777777777777
(2+3j)/(5+3j)
(0.5588235294117647+0.2647058823529412j)
True/False
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    True/False
ZeroDivisionError: division by zero
False/True
0.0
```

- **Floor Division(//):** It is used to get the quotient as the output. And the output will be displayed in the int format but float is exception.

  Syntax: OP1 // OP2

```
10//2
5
2.3//2
1.0
2//2.3
0.0
2.3//2.3
1.0
(2+3j)//(4+7j)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    (2+3j)//(4+7j)
TypeError: unsupported operand type(s) for //: 'complex' and 'complex'
True//True
1
```

- **Modulus(%):** It will consider remainder as the output.

  Syntax: OP1 % OP2

```
13%3
1
4.5%2.7
1.7999999999999998
(2+3j)%(4j)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    (2+3j)%(4j)
TypeError: unsupported operand type(s) for %: 'complex' and 'complex'
True%True
0
```

- **Power(**): --- It will work only on SVDT**
  It is going to specify the power of the given value.

```
2**3
8
4.5**2
20.25
(1+2j)**6
(117+44j)
True*7
7
True**7
1
4.3**6.2
8462.61190474804
(1+2j)**4.5
(9.963495519579647-36.03153123263964j)
16**(1/3)
2.5198420997897464
5**True
5
True**100
1
```

```
True**False
1
3.3453454**0
1.0
```

Day-13

2) Logical operator:

- **Logical AND:**
  --- If any one operand is False, The result is False.

  Syntax: OP1 and OP2

  Note:
  - If OP1 is False,
    output = OP1
  - If OP1 is True,
    output = OP2

```
10 and 22
22
0.0 and 9.8
0.0
2 and 0j
0j
[9.7] and 3.4
3.4
[] and ()
[]
2 and 0.0 and 5.6
0.0
```

- **Logical OR:**
  --- If any one operand is True, The result is True.

  Syntax: OP1 or OP2

  Note:
  - If OP1 is False,
    output = OP2
  - If OP1 is True,
    output = OP1

```
'tea' or 'coffee'
'tea'
6 or 9
6
0 or 123
123
[] or ''
''
78 or 34 or 5
78
67 and 23 or 89 and 45
23
```

- **Logical NOT: --- Output will be in the form of boolean**
  --- It gives negation of values.

Syntax: not(OP)

Note:

- If OP is False
  output = True
- If OP is True,
  output = False

```
not(7)
False
not(())
True
not([''])
False
```

3) Relational Operator / Comparision Operator: --- Output will be in the form of boolean.

- **Equal to (==):** It is used to check whether both the operands are same or not.

  Syntax: OP1 == OP2

  For SVDT: --- It will only check the value.

```
10 == 10
True
10 == 10.0
True
True == 1
True
3.4 == 4.5
False
```

  For MVDT: --- It will check both value and datatype.

```
10 == 10
True
10 == 10.0
True
True == 1
True
3.4 == 4.5
False
'hi' == 'hi'
True
[10,20] == (10,20)
False
{1,2,3} == {3,1,2}
True
[1,2,3] == (3,1,2)
False
[1,2,3] == [3,1,2]
False
{'a':10} == {'a':20}
False
{'a':10} == {'a':10}
True
```

- **Greater than (>): --- It will not support for complex and dictionary**
  --- Used to check whether the first operand is greater than the second operand or not.

Syntax: OP1 > OP2

For SVDT:

```
25 > 12
True
3.51 > 3.5
True
(2+3j) > (4+7j)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    (2+3j) > (4+7j)
TypeError: '>' not supported between instances of 'complex' and 'complex'
4j > 2j
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    4j > 2j
TypeError: '>' not supported between instances of 'complex' and 'complex'
1 > True
False
```

For MVDT:

- **If val1 > v1**
  output = True
- **If val1 < v1**
  output = False
- **If val1 == v1**
  Then it will go and check the next value.

For string, it will consider the ASCII values of characters.
ASCII --- American Standard Code for Information Interchange.

For set, it will check whether all the values of OP2 is present in OP1 or not.
If all the values of OP2 present in OP1 then len(OP1) should be greater then only it will return True.

```
ord('A')
65
ord('Z')
90
ord('a')
97
ord('z')
122
'python' > 'pycharm'
True
'ABC' > 'abc'
False
[10,20,30] > [10,20,10]
True
(10,20,30) > (10,20,10)
True
{10,20,30} > {1,2,3}
False
{10,20,30,1,2,3} > {1,2,3}
True
{10,20,30} > {10,20,30}
```

```
False
{10,20,30,40} > {10,20,30}
True
{10,20,30} > {20,10,30,40}
False
{10,20,30} >= {10,20,30}
True
{'a':10} > {'a':5}
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    {'a':10} > {'a':5}
TypeError: '>' not supported between instances of 'dict' and 'dict'
```

- **Lesser than (<): --- It will not support for complex and dictionary**
  --- Used to check whether the first operand is lesser than the second operand or not.

  Syntax: OP1 < OP2

  For SVDT:

```
20<50
True
True<1
False
3.4<3.33
False
```

  For MVDT:

  - **If val1 < v1**
    output = True
  - **If val1 > v1**
    output = False
  - **If val1 == v1**
    Then it will go and check the next value.

  For string, it will consider the ASCII values of characters.
  ASCII --- American Standard Code for Information Interchange.

  For set, it will check whether all the values of OP1 is present in OP2 or not.
  Then, If all the values of OP1 present in OP2 then len(OP2) should be greater then only it will return True.

```
'data' < 'science'
True
ord('d')
100
ord('s')
115
[12,35,56,78] < [12,35,12]
False
(12,36,52) < (12,36)
False
{True,89,25} < {1,56,89,False,25}
True
{True,89,25} < {1,89,False,25}
True
```

```
{True,89,25} < {1,89,25}
False
```

Day-14

- **Greater than or equal to (>=): --- It will not support for complex and dictionary**
  --- Used to check whether the first operand is greater than or equal to the second operand or not.

  Syntax: OP1 >= OP2

  ### For SVDT:

```
10>=10
True
3.4>=5.6
False
True>=False
True
```

  ### For MVDT:

```
'great' >= 'grapes'
True
ord
<built-in function ord>
ord('e')
101
ord('a')
97
[10,20,30]>=[10,20,30]
True
(12,3,7,6,True) >= (12,3,9)
False
{1,2,3} >=  {2,1,3}
True
```

- **Lesser than or equal to (>=): --- It will not support for complex and dictionary**
  --- Used to check whether the first operand is lesser than or equal to the second operand or not.

  Syntax: OP1 <= OP2

  ### For SVDT:

```
10<=10.0
True
3.4<=7.2
True
True<=False
False
```

  ### For MVDT:

```
'python' <= 'java'
False
[3,4,5,7]<=[3,4,5,8]
True
(1,2,3,True)<=(1,2,3,1)
True
{3,'hello',8,5}<={24,56,3,'hello',5,8}
```

True

- **Not equal to (!=):**
  --- It is used to check if operand 1 is not equal to operand 2 or not.

  Syntax: OP1 != OP2

  For SVDT:

```
10!=10
False
2.3!=3.2
True
(2+3j)==(3j+2)
True
(3j+2)
(2+3j)
(2+3j)!=(4j+2)
True
True != False
True
```

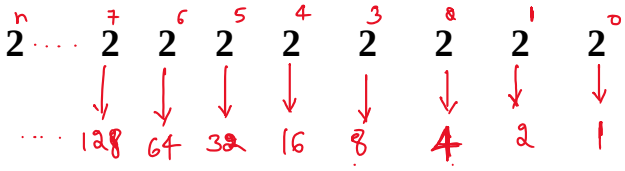
  For MVDT:

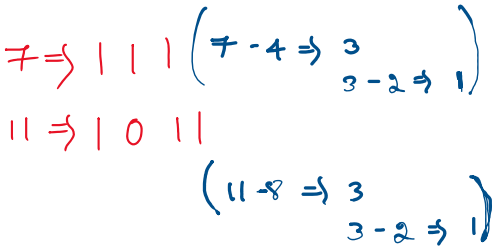
```
'happy'!='happ'
True
[10,20,30]!=[10,20,30]
False
(1,2,3)!=(True,True+True,True+True+True)
False
{1,'hello',True}!={True,True,'hello'}
False
```

4) **Bitwise Operator: --- It is used only for integers**
   --- It will consider the binary values to perform operations bit by bit.

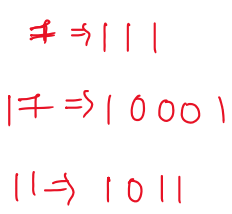
   We can get binary values by 3 methods,

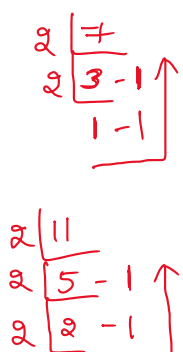
- **Binary Scale Method**

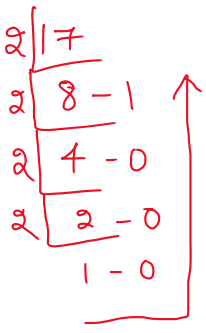


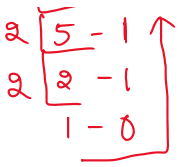


- **Divide by 2 method**





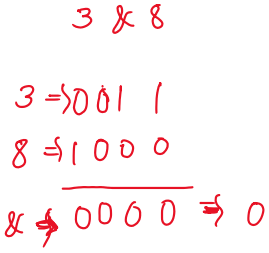


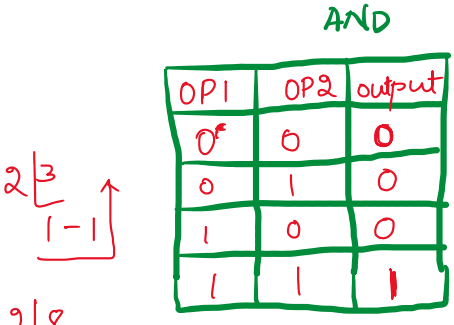


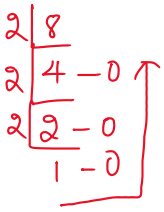
- **Using inbuilt function ,**
  Syntax: bin(val)

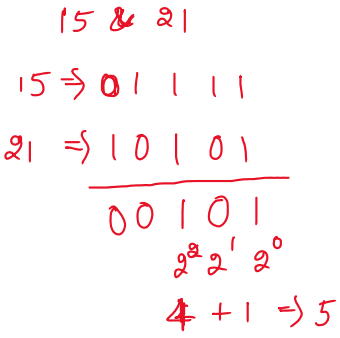
```
bin(7)
'0b111'
bin(17)
'0b10001'
bin(11)
'0b1011'
```

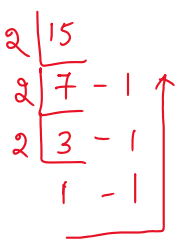
- **Bitwise AND(&):**
  Syntax: OP1 & OP2

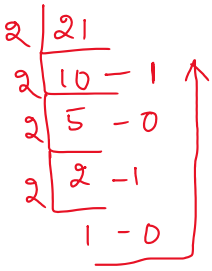








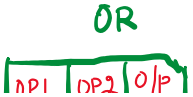




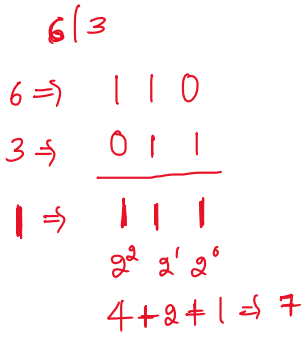
Practical proof:

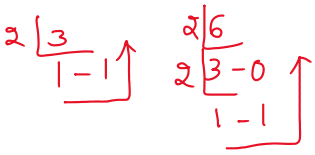
```
3 & 8
0
15 & 21
5
```

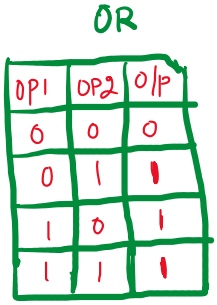
- **Bitwise OR(|):**
  Syntax: OP1 | OP2

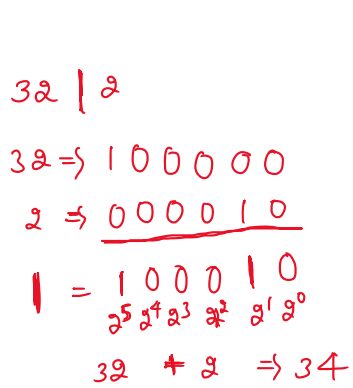


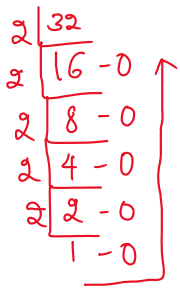
- **Bitwise OR(|):**
  Syntax: OP1 | OP2







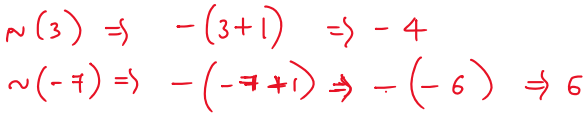




Practical proof:

```
6 | 3
7
32 | 2
34
```

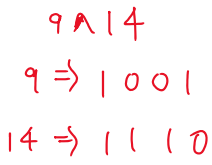
- **Bitwise NOT(~):**
  Syntax: ~(OP) results will be displayed like --- -(OP+1)

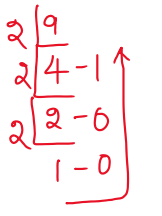


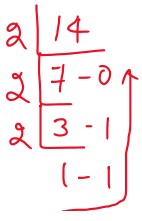
Practical proof:

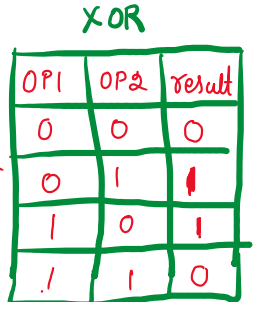
```
~(3)
-4
~(-7)
6
```

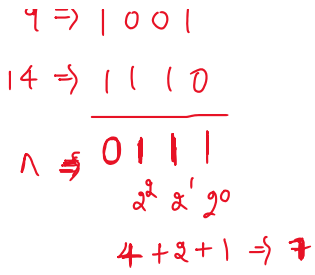
- **Bitwise XOR(^):**
  Syntax: OP1 ^ OP2

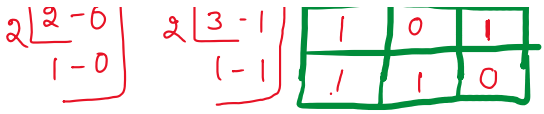








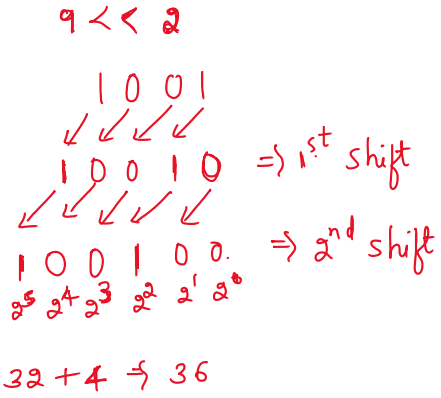




Practical proof:

```
9 ^ 14
7
```

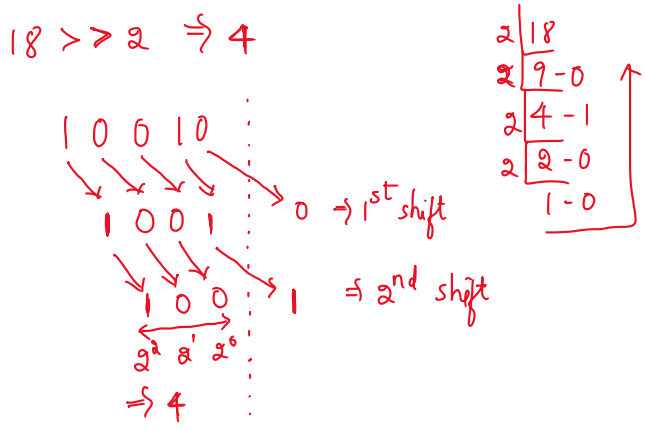
- **Bitwise left shift(<<):**
  Syntax: OP << n **n ---- number of shifts**



Practical proof:

```
9 << 2
36
9 << 1
18
9 << 5
288
```

- **Bitwise right shift(>>):**
  Syntax: OP >> n **n ---- number of shifts**



Practical proof:

```
18 >> 2
4
```

Day-15

5) **Assignment Operator (=):**
---- It is used to assign the value to a varaible.

a=a + b --- a += b
a=a - b --- a -= b
a = a * b --- a*=b
a=a/b --- a/=b
a=a//b --- a//=b
a=a%b --- a%=b
a=ab --- a**=b**

6) **Membership Operator:**
--- It is used to check if the value is present inside the collection or not.

- in --- Tells values present in the collection. It will return True if the values are present in the collection else return False.
- not in --- Tells values not present in the collection. It will return True if the values are not present in the collection else return False.

Practical proof:

```
'hi' in 'hi-bye'
True
'hb' in 'hi-bye'
False
1 in 123
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    1 in 123
TypeError: argument of type 'int' is not iterable
1 in '123'
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    1 in '123'
TypeError: 'in <string>' requires string as left operand, not int
'1' in '123'
True
[10] in [10,20,30]
False
[10] in [[10],20,30]
True
10 in [10,20,30]
True
10, in (10,20,30)
SyntaxError: invalid syntax
(10,) in (10,20,30)
False
(10) in (10,20,30)
True
10 not in {'a':10, 'b':20}
```

```
True
10 not in {10:10, 'b':20}
False
1,2,4 in {1,2,4}
(1, 2, True)
1,2,4 in {12,4}
(1, 2, True)
1,2,4 in {(12,),4}
(1, 2, True)
'' in 'hello'
True
[] in [10,20,30]
False
[] in [10,[],20,30]
True
() in (10,20,30)
False
() in (10,(),20,30)
True
a='hello'
a[3]
'l'
a=(10,20,30)
a[1]
20
```

7) Identity Operator:

--- It is used to check whether both the variables are pointing to the same address or not.

- is ----- It will return True if both the variables are pointing to the same address or else return False.
- is not ----- It will return True if both the variables are not pointing to the same address or else return False.

Practical proof:

```
a=10
b=20
c=10
a is b
False
a is c
True
a is not b
True
a is not c
False
```

Input and Output Statements:

- **Input Statement :** We should never allow the user to modify the code, instead just allow them to access it.

Syntax:

```
var = input('Enter your message')
```

Practical proof:

```
a=input('Enter your input: ')
print(a,type(a))
```

Note:

- input() function will take input by default in the form of string.
- If u want actual datatype , then use typecasting (only for SVDT)

```
=int(input('Enter your input: '))
print(a,type(a))

a=float(input('Enter your input: '))
print(a,type(a))

a=complex(input('Enter your input: '))
print(a,type(a))

a=bool(input('Enter your input: '))
print(a,type(a))
```

- When we want to do typecasting for MVDT use the ,
  eval function --- It will consider the data according to its type

```
a=eval(input('Enter your input: '))
print(a,type(a))
```

- **Output Statement:**
  --- Whenever we want to display the output we use output statement.

Syntax:

print(val1,val2,…….valn, sep=' ', end='\n')

default parameters

Practical proof:

print(10,20,30,40)	------	10 20 30 40
print(10,20,30,40,sep='@')	------	10@20@30@40
print(10,20,30,40,end='%')	------	10 20 30 40%

Programs:

```
# WAP to add 2 numbers.
'''
a=int(input('Enter the num1: '))
b=int(input('Enter the num2: '))
print(a+b)'''

# WAP to print the square of the number.
'''
a=int(input('Enter the num: '))
print(a**2)'''

# WAP to extract last character from the string.
'''
s=input('Enter the string: ')
print(s[-1])'''
```

```
# WAP to get the values from the even index in the tuple.
'''
t=eval(input('Enter the tuple: '))
print(t[::2])'''

# WAP to extract the last digit from the integer
'''
a=int(input('Enter the num: '))
b=str(a)
print(b[-1])
print(a%10)'''
```